Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию
для многоядерных компьютерных систем

Образовательный комплекс "Введение в методы параллельного программирования"

Раздел " Параллельные методы решения систем линейных уравнений"

Лабораторная работа 3: Параллельные методы решения систем линейных уравнений

Разработчик: Е.А.Козинов

Нижний Новгород
2007

Содержание

Удалено: 16

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

В данной лабораторной работе рассматривается один из прямых методов решения систем линейных уравнений – метод Гаусса и его параллельное обобщение.

Цель лабораторной работы

Целью данной лабораторной работы является разработка параллельной программы, которая выполняет решение системы линейных уравнений методом Гаусса. Выполнение лабораторной работы включает:

- Упражнение 1 – Постановка задачи решения системы линейных уравнений,
- Упражнение 2 – Изучение последовательного алгоритма Гаусса решения систем линейных уравнений,
- Упражнение 3 – Реализация последовательного алгоритма Гаусса,
- Упражнение 4 – Разработка параллельного алгоритма Гаусса,
- Упражнение 5 - Реализация параллельного алгоритма Гаусса решения систем линейных уравнений.

Примерное время выполнения лабораторной работы: **90 минут**.

При выполнении лабораторной работы предполагается знание раздела 4 "Параллельное программирование с использованием OpenMP", раздела 6 "Принципы разработки параллельных методов" и раздела 9 "Параллельные методы решения систем линейных уравнений" учебных материалов курса. Кроме того, предполагается, что выполнена лабораторная работа 1 "Параллельные алгоритмы матрично-векторного умножения" и лабораторная работа 2 "Параллельные алгоритмы матричного умножения".

Удалено:

Упражнение 1 – Постановка задачи решения системы линейных уравнений

Линейное уравнение с n неизвестными x_0, x_1, ..., x_{n-1} может быть определено при помощи выражения

$$a_0x_0 + a_1x_1 + \ldots + a_{n-1}x_{n-1} = b \qquad (3.1)$$

где величины a_0, a_1, ..., a_{n-1} и b представляют собой постоянные значения.

Множество n линейных уравнений

$$\begin{aligned} a_{0,0}x_0 \quad &+a_{0,1}x_1 \quad &+\ldots+a_{0,n-1}x_{n-1} \quad &= b_0 \\ a_{1,0}x_0 \quad &+a_{1,1}x_1 \quad &+\ldots+a_{1,n-1}x_{n-1} \quad &= b_1 \\ \ldots & & & \\ a_{n-1,0}x_0 \quad &+a_{n-1,1}x_1 \quad &+\ldots+a_{n-1,n-1}x_{n-1} \quad &= b_{n-1} \end{aligned} \qquad (3.2)$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (*матричном*) виде система может представлена как

$$Ax = b\ ,$$

где $A=(a_{i,j})$ есть вещественная матрица размера $n \times n$, а вектора b и x состоят из n элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы A и вектора b обычно понимается нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

Упражнение 2 - Изучение последовательного алгоритма Гаусса решения систем линейных уравнений

Метод Гаусса является широко известным *прямым* алгоритмом решения систем линейных уравнений, для которых матрицы коэффициентов являются *плотными*. Если система линейных уравнений является *невырожденной*, то метод Гаусса гарантирует нахождение решения с погрешностью,

определяемой точностью машинных вычислений. Основная идея метода состоит в приведении матрицы A посредством эквивалентных преобразований (не меняющих решение системы (3.2)) к треугольному виду, после чего значения искомых неизвестных может быть получено непосредственно в явном виде.

В упражнении дается общая характеристика метода Гаусса, достаточная для начального понимания алгоритма и позволяющая рассмотреть возможные способы параллельных вычислений при решении систем линейных уравнений.

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование *эквивалентных*). К числу таких преобразований относятся:

- Умножение любого из уравнений на ненулевую константу,
- Перестановка уравнений,
- Прибавление к уравнению любого другого уравнения системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – *прямой ход* метода Гаусса – исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду

$$U x = c ,$$

где матрица коэффициентов получаемой системы имеет вид

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}.$$

На *обратном ходе* метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д.

Прямой ход алгоритма Гаусса

Прямой ход метода Гаусса состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений. На итерации i, $0 \le i < n-1$, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i (т.е. $i < k \le n-1$). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу (a_{ki}/a_{ii}), с тем, чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki} / a_{ii}) \cdot a_{ij}, \\ b'_k &= b_k - (a_{ki} / a_{ii}) \cdot b_i, \end{aligned} \quad i \le j \le n-1, i < k \le n-1, 0 \le i < n-1 \qquad (3.3)$$

(следует отметить, что аналогичные вычисления выполняются и над элементами вектора b).

Поясним выполнение прямого хода метода Гаусса на примере системы линейных уравнений вида:

$$\begin{aligned} x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ 2x_0 &+ 7x_1 &+ 5x_2 &= 18 . \\ x_0 &+ 4x_1 &+ 6x_2 &= 26 \end{aligned}$$

На первой итерации производится исключение неизвестной x_0 из второй и третьей строки. Для этого из этих строк нужно вычесть первую строку, умноженную соответственно на 2 и 1. После этих преобразований система уравнений принимает вид:

$$\begin{aligned} x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ & x_1 &+ x_2 &= 16 . \\ & x_1 &+ 4x_2 &= 25 \end{aligned}$$

В результате остается выполнить последнюю итерацию и исключить неизвестную x_1 из третьего уравнения. Для этого необходимо вычесть вторую строку и в окончательной форме система имеет следующий вид:

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad + x_2 \quad &= \quad 16. \\ 3x_2 \quad &= \quad 9 \end{aligned}$$

На рис. 3.1 представлена общая схема состояния данных на *i*-ой итерации прямого хода алгоритма Гаусса. Все коэффициенты при неизвестных, расположенные ниже главной диагонали и левее столбца *i*, уже являются нулевыми. На *i*-ой итерации прямого хода метода Гаусса осуществляется обнуление коэффициентов столбца *i*, расположенных ниже главной диагонали, путем вычитания строки *i*, умноженной на нужную ненулевую константу. После проведения *(n-1)* подобной итерации матрица, определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

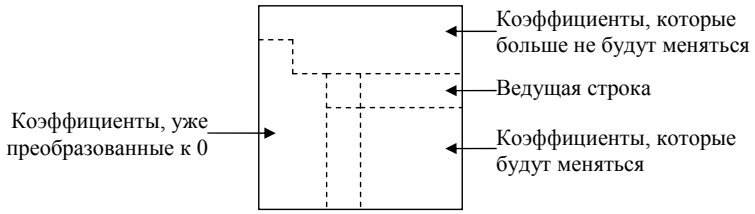

Рис. 3.1. Итерация прямого хода алгоритма Гаусса

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*. Как можно заметить, выполнение вычислений является возможным только, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент $a_{i,i}$ имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Возможный способ избежать подобной проблемы может состоять в следующем – при выполнении каждой очередной итерации прямого хода метода Гаусса следует определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т.е.

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование *метода главных элементов*).

Вычислительная сложность прямого хода алгоритма Гаусса с выбором ведущей строки имеет порядок $O(n^3)$.

Обратный ход алгоритма Гаусса

После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д. В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$\begin{aligned} & x_{n-1} = b_{n-1} / a_{n-1,n-1}, \\ & x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \;\; i = n-2, n-3, ..., 0 \end{aligned} \qquad (3.4)$$

Поясним, как и ранее, выполнение обратного хода метода Гаусса на примере рассмотренной в предыдущем подразделе системы линейных уравнений

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad + x_2 \quad &= \quad 16. \\ 3x_2 \quad &= \quad 9 \end{aligned}$$

Из последнего уравнения системы можно определить, что неизвестная x_2 имеет значение 3. В результате становится возможным разрешение второго уравнения и определение значение неизвестной x_1=13, т.е.

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1 \\ x_1 \quad &= \quad 13 \\ x_2 \quad &= \quad 3 \end{aligned}.$$

На последней итерации обратного хода метода Гаусса определяется значение неизвестной x_0, равное -44.

С учетом последующего параллельного выполнения можно отметить, учет получаемых значений неизвестных может выполняться сразу во всех уравнениях системы (и эти действия могут выполняться в уравнениях одновременно и независимо друг от друга). Так, в рассматриваемом примере после определения значения неизвестной x_2 система уравнений может быть приведена к виду

$$\begin{aligned} x_0 \quad +3x_1 \quad &= \quad -5 \\ x_1 \quad &= \quad 13 \\ x_2 \quad &= \quad 3 \end{aligned}$$

Вычислительная сложность обратного хода алгоритма Гаусса составляет $O(n^2)$.

Упражнение 3 – Реализация последовательного алгоритма Гаусса

При выполнении этого упражнения необходимо реализовать последовательный алгоритм Гаусса решения систем линейных уравнений. Начальный вариант будущей программы представлен в проекте *SerailGauss*, который содержит часть исходного кода и в котором заданы необходимые параметры проекта. В ходе выполнения упражнения необходимо дополнить имеющийся вариант программы операциями ввода размера матриц, задание исходных данных, реализации алгоритма Гаусса и вывода результатов.

Задание 1 – Открытие проекта SerialGauss

Откройте проект **SerialGauss**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню **File** выполните команду **Open→Project/Solution,**
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\SerialGauss,**
- Дважды щелкните на файле **SerialGauss.sln** или выбрав файл выполните команду **Open.**

После открытия проекта в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialGauss.cpp**, как это показано на рис. 3.2. После этих действий код, который предстоит в дальнейшем расширить, будет открыт в рабочей области Visual Studio.

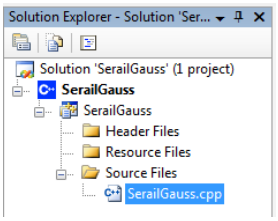

Рис. 3.2. Открытие файла **SerialGauss.cpp**

В файле **SerialGauss.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции *main*. Эта заготовка содержит объявление переменных и вывод на печать начального сообщения программы.

Рассмотрим переменные, которые используются в основной функции (*main*) нашего приложения. Первые две из них (*pMatrix* и *pVector*) – это, соответственно, матрица системы линейных уравнений и вектор правых частей системы. Третья переменная *pResult* – вектор, который должен быть получен в результате решения системы линейных уравнений. Переменная *Size* определяет размер матрицы и векторов.

Отформатировано: интервал
После: 6 пт

```
double* pMatrix;  // The matrix of linear system
double* pVector;  // The right parts of the linear system
double* pResult;  // The result vector
int Size;         // Sizes of the initial matrix and the vector
```

Как и в предыдущих лабораторных работах, для хранения матрицы используется одномерный массив, в котором матрица хранятся построчно. Таким образом, элемент, расположенный на пересечении *i*-ой строки и *j*-ого столбца матрицы, в одномерном массиве имеет индекс *i*Size+j*.

Программный код, который следует за объявлением переменных, это вывод начального сообщения и ожидание нажатия любой клавиши перед завершением выполнения приложения:

```
  printf("Serial Gauss algorithm for solving linear systems\n");
  getch();
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна Visual Studio появилось сообщение "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

Сразу после запуска кода, в командной консоли появится сообщение: "`Serial Gauss algorithm for solving linear systems `". Для того, чтобы завершить выполнение программы, нажмите любую клавишу.

Задание 2 – Ввод размеров матрицы и вектора

Для задания исходных данных последовательного алгоритма Гаусса решения системы линейных уравнений реализуем функцию *ProcessInitialization*. Эта функция предназначена для определения размера матрицы и векторов, выделения памяти для исходных матрицы *pMatrix* и вектора *pVector*, и вектора-результата *pResult*, а также для задания значений элементов исходных объектов. Значит, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size);
```

На первом этапе необходимо определить размер матриц (задать значение переменной *Size*). В тело функции *ProcessInitialization* добавьте выделенный фрагмент кода:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  // Setting the size of the matrix and the vector
  printf("\nEnter the size of the matrix and the vector: ");
  scanf("%d", &Size);
  printf("\nChosen size = %d", Size);
}
```

Пользователю предоставляется возможность ввести размер матриц, который затем считывается из стандартного потока ввода *stdin* и сохраняется в целочисленной переменной *Size*. Далее печатается значение переменной *Size* (рис. 3.3).

После строки, выводящей на экран приветствие, добавьте вызов функции инициализации процесса вычислений *ProcessInitialization* в тело основной функции последовательного приложения:

```
void main() {
  double* pMatrix;  // The matrix of the linear system
  double* pVector;  // The right parts of the linear system
  double* pResult;  // The result vector
  int Size;         // The sizes of the initial matrix and the vector
  time_t start, finish;
  double duration;

  printf ("Serial Gauss algorithm for solving linear systems \n");
  ProcessInitialization(pMatrix, pVector, pResult, Size);
  getch();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что значение переменной *Size* задается корректно.

Отформатировано: интервал После: 6 пт

Отформатировано: интервал Перед: 6 пт

Отформатировано: интервал После: 6 пт

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: французский (Франция)

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: французский (Франция)

Отформатировано: интервал Перед: 6 пт

Отформатировано: интервал После: 6 пт

Отформатировано: интервал Перед: 6 пт, После: 6 пт

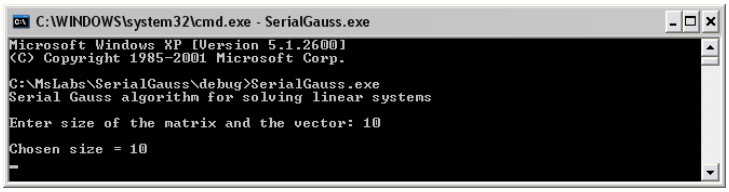

Рис. 3.3. Задание размера объектов

Как и при выполнении предыдущих лабораторных работ, выполним контроль правильности ввода. Организуем проверку размера и, в случае ошибки (заданный размер является нулевым или отрицательным), продолжим запрашивать размер матриц до тех пор, пока не будет введено положительное число. Для реализации такого поведения поместим фрагмент кода, который производит ввод размера матриц, в цикл с постусловием:

```
  // Setting the size of the matrix and the vector
  do {
    printf("\nEnter the size of the matrix and the vector: ");
    scanf("%d", &Size);
    printf("\nChosen size = %d\n", Size);

    if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);
```

Снова скомпилируйте и запустите приложение. Попытайтесь ввести неположительное число в качестве размера объектов. Убедитесь в том, что ошибочные ситуации обрабатываются корректно.

Задание 3 – Ввод данных

Функция инициализации процесса вычислений должна осуществлять также выделение памяти для хранения объектов (добавьте выделенный код в тело функции *ProcessInitialization*):

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
double* &pResult, int &Size) {
  // Setting the size of the matrix and the vector
  do {
    <…>
  }
  while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];
}
```

Далее необходимо задать значения элементов матрицы системы линейных уравнений *pMatrix* и вектора правых частей *pVector*. Заметим, что матрица системы линейных уравнений не может быть задана произвольным образом. Решение системы линейных уравнений существует только в случае, когда матрица системы линейных уравнений *невырожденная* (то есть для нее существует обратная матрица). Проводить проверку матрицы, сгенерированной случайным образом, на невырожденность нецелесообразно (это слишком "дорогостоящая" операция). Поэтому реализуем функции генерации исходных данных таким образом, чтобы матрица изначально являлась невырожденной. Будем генерировать нижнюю треугольную матрицу, то есть матрицу, у которой все ненулевые элементы, расположены либо на главной диагонали, либо ниже ее. Для задания значений элементов матрицы *pMatrix* и вектора *pVector* реализуем функцию *DummyDataInitialization*. Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  for (i=0; i<Size; i++) {
```


Отформатировано: интервал После: 6 пт

Отформатировано: интервал Перед: 6 пт

Отформатировано: интервал После: 6 пт

Отформатировано: французский (Франция)

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

```
    pVector[i] = i+1;
    for (j=0; j<Size; j++) {
      if (j <= i)
        pMatrix[i*Size+j] = 1;
      else
        pMatrix[i*Size+j] = 0;
     }
  }
}
```

Как видно из представленного фрагмента кода, данная функция осуществляет задание элементов матрицы и вектора простым образом: значения всех элементов матрицы *pMatrix* , расположенные выше главной диагонали, равны 0, остальные элементы равны 1. Вектор *pVector* состоит из последовательных целых положительных чисел от 1 до *Size*. То есть в случае, когда пользователь выбрал размер объектов, например, равный 4, будут определены следующие матрица и вектор:

$$pMatrix = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \; pVector = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

Вызов функции *DummyDataInitialization* необходимо выполнить после выделения памяти внутри функции *ProcessInitialization*:

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, int &Size) {
  <…>
  // Memory allocation
  <…>

  // Initialization of the matrix and the vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
}
```

Для контроля ввода данных воспользуемся функциями форматированного вывода объектов *PrintMatrix* и *PrintVector*, которые были разработаны при выполнении лабораторной работы 1 и текст которых уже имеется в проекте (подробнее о функциях *PrintMatrix* и *PrintVector* см. задание 3 упражнение 2 лабораторной работы 1). Добавим вызов этих функций для печати объектов *pMatrix* и *pVector* в основную функцию приложения:

```
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ввод данных происходит по описанным правилам (рис. 3.4). Выполните несколько запусков приложения, задавайте различные размеры матриц.

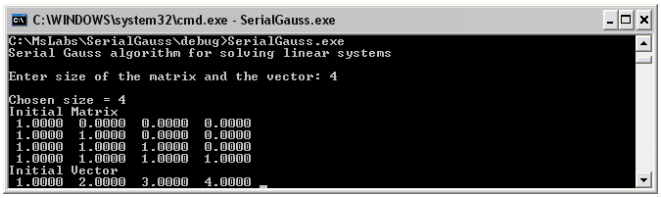

Рис. 3.4. Результат работы программы при завершении задания 3

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: французский (Франция)

Удалено:

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Следует отметить, что если матрица системы линейных уравнений и вектор правых частей задаются по описанным выше правилам, то такая система имеет простое решение, все элементы искомого вектора *pResult* должны быть равны 1.

Реализуем еще одну функцию генерации исходных данных, в которой по-прежнему будет задаваться нижняя треугольная матрица, но элементы этой матрицы и вектора правых частей будут определятся с помощью датчика случайных чисел (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of the matrix and the vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
    for (j=0; j<Size; j++) {
      if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
      else
        pMatrix[i*Size+j] = 0;
    }
  }
}
```

Замените вызов функции простой генерации исходных данных *DummyDataInitialization* вызовом функции случайной генерации *RandomDataInitialization*. Скомпилируйте и запустите приложение. Убедитесь в том, что данные задаются согласно описанным правилам.

Задание 4 – Завершение процесса вычислений

Перед выполнением матрично-векторного умножения сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию *ProcessTermination*. Память выделялась для хранения исходных матрицы *pMatrix* и вектора *pVector*, а также для хранения вектора - результата решения системы линейных уравнений *pResult*. Следовательно, эти объекты необходимо передать в функцию *ProcessTermination* в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult) {
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}
```

Вызов функции *ProcessTermination* необходимо выполнить непосредственно перед завершением программы:

```
  // Memory allocation and definition of the objects elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // Matrix and vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Process termination
  ProcessTermination(pMatrix, pVector, pResult);
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

Задание 5 – Реализация прямого хода метода Гаусса

Выполним теперь разработку основной вычислительной части программы. Для решения системы линейных уравнений при помощи последовательного алгоритма Гаусса реализуем функцию

Отформатировано: интервал Перед: 6 пт

Отформатировано: интервал После: 6 пт

Отформатировано: интервал Перед: 6 пт

Отформатировано: интервал После: 6 пт

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: интервал Перед: 6 пт

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

SerialResultCalculation, которая принимает на вход исходные матрицу *pMatrix* и вектор *pVector*, размер этих объектов *Size*, а также указатель на вектор-результат *pResult*.

В соответствии с алгоритмом, изложенным в упражнении 2, выполнение алгоритма Гаусса состоит из двух этапов: прямого хода метода Гаусса и обратного хода метода Гаусса. На этапе выполнения прямого хода метода Гаусса система линейных уравнений путем эквивалентных преобразований приводится к верхнему треугольному виду. Для выполнения этого этапа реализуем функцию *SerialGaussianElimination*. При выполнении обратного хода алгоритма Гаусса определяются значения искомого вектора путем приведения матрицы к диагональному виду. Для выполнения этого этапа реализуем функцию *SerialBackSubstitution*. Таким образом, код функции *SerialResultCalculation* должен выглядеть следующим образом:

```
// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);
}
```

В данном задании будет выполнена реализация прямого хода метода Гаусса. Обратный ход метода Гаусса будет выполнен в следующем задании лабораторной работы.

Прямой ход метода Гаусса путем эквивалентных преобразований приводит матрицу системы линейных уравнений к верхнему треугольному виду. На каждой итерации выполняемых преобразований для выбора ведущей строки применяют *метод главных элементов* (см. упражнение 2), в соответствии с которым в качестве ведущей выбирается строка, содержащая максимальный по абсолютному значению элемент очередного столбца матрицы.

Для запоминания порядка выбора ведущих строк введем массив *pSerialPivotPos*, в *i*-ом элементе которого будем хранить номер строки, которая была выбрана в качестве ведущей при выполнении *i*-ой итерации прямого хода алгоритма Гаусса. Кроме того, определим еще один дополнительный массив – *pSerialPivotIter*, в каждом элементе которого *pSerialPivotIter*[*i*] будем хранить номер итерации, на которой строка с номером *i* выбиралась в качестве ведущей. Изначально массив *pSerialPivotIter* заполним элементами, равными -1 (т.е. значение -1 в элементе массива *pSerialPivotIter*[*i*] означает, что строка с номером *i* еще не выбиралась в качестве ведущей).

Объявим соответствующие массивы как глобальные переменные, выделим память для этих массивов перед началом выполнения функции *GaussianElimination*, освободим выделенную память после завершения выполнения обратного хода метода Гаусса (функции *BackSubstitution*):

```
int* pSerialPivotPos;  // The number of pivot rows selected at the
                       // iterations
int* pSerialPivotIter; // The iterations, at which the rows were pivots

// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pSerialPivotPos  = new int [Size];
  pSerialPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pSerialPivotPos;
  delete [] pSerialPivotIter;
}
```

Отформатировано: интервал После: 6 пт

Отформатировано: интервал Перед: 6 пт

Отформатировано: Шрифт: не курсив

Отформатировано: Шрифт: не курсив

Отформатировано: Шрифт: не курсив

Отформатировано: Шрифт: не курсив

Отформатировано: интервал После: 6 пт

Согласно вычислительной схеме прямого хода алгоритма Гаусса, на каждой итерации необходимо определить ведущую строку матрицы, то есть строку, которая содержит максимальный по абсолютной величине элемент в столбце с номером, равным номеру текущей итерации, среди тех строк, которые ранее не были выбраны в качестве ведущих. Номер ведущей строки запоминается в переменной *Pivot* и записывается в соответствующий элемент массива *pSerialPivotPos*. Кроме того, значение элемента массива *pSerialPivotIter*, соответствующего выбранной строке, устанавливается равным номеру текущей итерации.

Реализуем функцию *FindPivotRow* для выбора ведущей строки. В качестве аргументов этой функции необходимо передать матрицу системы линейных уравнений *pMatrix*, размер матрицы *Size* и номер текущей итерации *Iter*. Эта функция должна просмотреть все строки, которые ранее не были выбраны в качестве ведущих, выбрать среди них ту, которая содержит максимальный элемент в позиции *Iter*, и вернуть номер выбранной строки:

```
// Function for finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1; // The index of the pivot row
  double MaxValue =  0; // The value of the pivot element
  int i;             // Loop variable

  // Choose the row, that stores the maximum element
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
       (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
    }
  }
  return PivotRow;
}
```

Добавим вызов функции *FindPivotRow* в тело функции, выполняющей прямой ход метода Гаусса, запомним найденное значение в соответствующем элементе массива *pSerialPivotPos* и напечатаем номера выбираемых ведущих строк для проверки правильности вычислений:

```
// Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The Number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The Number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
  }
  printf ("Indices of the pivot rows: \n");
  for (int i=0; i<Size; i++)
    printf("%d ", pSerialPivotPos[i]);
}
```

В функции *SerialResultCalculation* закомментируйте вызов функции, выполняющей обратный ход метода Гаусса. Добавьте вызов функции *SerialResultCalculation* в главную функцию приложения:

```
void main() {
  <…>
  // Memory allocation and data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  <…>
  // Execution of Gauss algorithm
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
```


Отформатировано: интервал Перед: 6 пт

Отформатировано: интервал После: 6 пт

Удалено:

Удалено: F

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: интервал Перед: 6 пт

```
  getch();
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что ведущие строки выбираются правильно. При использовании функции *DummyDataInitialization* номера ведущих строк должны совпадать с номерами итераций, на которых эти строки выбирались. При использовании функции *RandomDataInitialization* общий вид результатов печати показан на рис. 3.5 (для наглядности значения ведущих элементов выделены красным цветом).

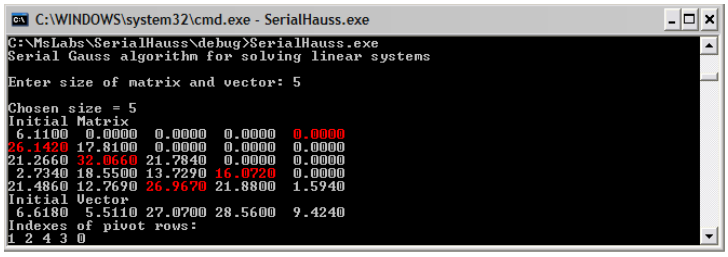

Рис. 3.5. Выбор ведущих строк: сначала выбираем строку, которая содержит максимальный элемент в первом столбце, затем среди всех строк, кроме второй, выбираем строку, содержащую максимальный элемент во втором столбце, и т.д.

Далее после выбора ведущих строк выполняется вычитание этих строк, умноженных на соответствующие множители, из всех строк, которые еще не выбирались в качестве ведущих, и таким образом зануляются элементы соответствующих столбцов. Для выполнения вычитания реализуем функцию *SerialEliminateColumns*, которая принимает на вход матрицу системы линейных уравнений *pMatrix*, вектор правых частей *pVector*, номер текущей ведущей строки *Pivot*, номер текущей итерации *Iter* и размер *Size*. Для всех строк матрицы *pMatrix* функция *EliminateRows* выполняет следующие действия: проверяет при помощи значений, записанных в массиве *pSerialPivotIter*, не была ли данная строка выбрана в качестве ведущей на одной из предшествующих итераций, и, если результат проверки отрицательный, то над этой строкой выполняются преобразования, согласно формуле (3.3):

```
// Column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}
```

Добавьте вызов функции *SerialColumnElimination* в код функции, выполняющей прямой ход алгоритма Гаусса. Вместо печати массива *pSerialPivotPos* распечатайте матрицу системы линейных уравнений *pMatrix*. Она должна быть приведена к верхнему треугольному виду с точностью до перестановки строк (то есть должна существовать возможность перестановки строк матрицы так, чтобы получилась верхняя треугольная матрица) (см. рис. 3.6).

```
// Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The number of the iteration of the gaussian
                  // elimination stage
  int PivotRow;   // The number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size,Iter);
    pSerialPivotPos[Iter] = PivotRow;
```


Отформатировано: английский (США)
Отформатировано: английский (США)
Отформатировано: английский (США)
Отформатировано: интервал Перед: 6 пт, После: 6 пт
Отформатировано: английский (США)
Отформатировано ... [1]
Отформатировано ... [2]
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано ... [3]
Отформатировано ... [4]
Отформатировано ... [5]
Отформатировано ... [6]
Отформатировано ... [7]
Отформатировано ... [8]
Отформатировано ... [9]
Отформатировано ... [10]
Отформатировано ... [11]
Отформатировано ... [12]
Отформатировано ... [13]
Отформатировано ... [14]
Отформатировано ... [15]
Отформатировано ... [16]

```
    pSerialPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
  // printf ("Indices of the pivot rows: \n");
  // for (int i=0; i<Size; i++)
  //  printf("%d ", pSerialPivotPos[i]);
  printf ("The matrix of the linear system after the elimination: \n");
  PrintMatrix(pMatrix, Size, Size);
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что прямой ход метода Гаусса выполняется правильно.

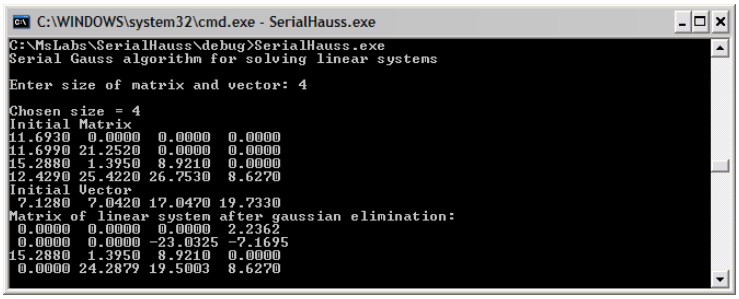

Рис. 3.6. Результат выполнения прямого хода алгоритма Гаусса

Задание 6 – Выполнение обратного хода алгоритма Гаусса

Для выполнения обратного хода алгоритма Гаусса реализуем функцию *SerialBackSubstitution*. На вход этой функции передадим матрицу системы линейных уравнений *pMatrix*, вектор правых частей *pVector*, вектор результата *pResult* и размер *Size*:

```
// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size);
```

Алгоритм обратного хода алгоритма Гаусса подробно описан в упражнении 2. Выполнение обратного хода начинается с той строки матрицы, которая была выбрана ведущей на последней итерации прямого хода. Номер этой строки можно узнать, обратившись к последнему элементу массива *pSerialPivotPos* (аналогично номер строки, которая была выбрана ведущей на предпоследней итерации прямого хода, хранится в предпоследнем элементе массива *pSerialPivotPos* и так далее). Используя эту строку можно вычислить один из элементов результирующего вектора, а затем, используя этот элемент, упростить остальные строки матрицы:

```
// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}
```

Удалите печать матрицы после выполнения прямого хода метода Гаусса. Для генерации исходных данных снова используйте метод *DummyDataInitialization*. Раскомментируйте вызов функции выполнения обратного хода метода Гаусса. Вызовите печать результирующего вектора после выполнения последовательного алгоритма Гаусса в основной функции приложения:

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: интервал После: 6 пт

Удалено:

Отформатировано: русский

Отформатировано: интервал Перед: 6 пт, После: 6 пт

```
void main() {
 <…>

  // The Execution of Gauss algorithm
  SerialResultCalculation(pMatrix, pVector, pResult, Size);

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Скомпилируйте и запустите приложение. Если алгоритм реализован верно, все элементы результирующего вектора должны быть равны 1 (рис. 3.7).

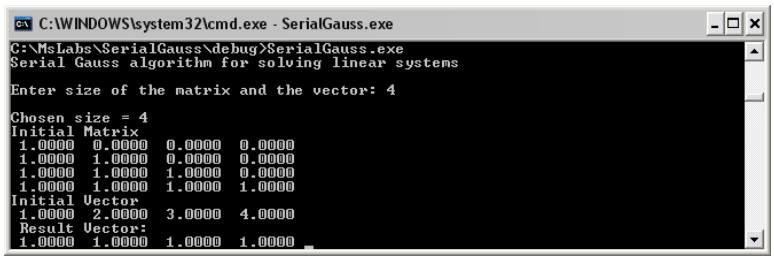

Рис. 3.7. Результат выполнения последовательного алгоритма Гаусса

Задание 7 – Проведение вычислительных экспериментов

Для последующего тестирования ускорения работы параллельного алгоритма необходимо провести эксперименты по вычислению времени выполнения последовательного алгоритма. Анализ времени выполнения алгоритма разумно проводить для достаточно больших размеров системы линейных уравнений. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел (функция *RandomDataInitialization*).

Для определения времени добавьте в получившуюся программу вызовы функций, позволяющие узнать время выполнения вычислений. Мы, как и ранее, будем пользоваться функцией, разработанной в лабораторной работе 1:

```
double GetTime();
```

Добавим в программный код вычисление и вывод времени выполнения метода Гаусса, для этого поставим замеры времени до и после вызова функции *SerialResultCalculation*:

```
  // The execution of Gauss algorithm
  start = GetTime();
  SerialResultCalculation(pMatrix, pVector, pResult, Size);
  finish = GetTime();
  duration = finish-start;

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the execution time of Gauss method
  printf("\n Time of execution: %f\n", duration);
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими объектами отключите печать исходных матрицы и вектора и печать результирующего вектора (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу:

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: английский (США)

Примечание [Г1]: Не нашел такую функцию в лаб. 1

Отформатировано: интервал Перед: 6 пт, После: 6 пт

Отформатировано: интервал Перед: 6 пт

Таблица 3.1. Время выполнения последовательного алгоритма Гаусса

Номер теста	Размер матрицы	Время работы (сек)
1	10	
2	100	
3	500	
4	1000	
5	1500	
6	2000	
7	2500	
8	3000	

Отформатировано: интервал После: 6 пт

Отформатировано: интервал Перед: 6 пт

В результате анализа выполняемых в методе Гаусса вычислений было показано, что теоретическое время выполнения последовательного алгоритма Гаусса может быть вычислено в соответствии с выражением (см. раздел 9 "Параллельные методы решения систем линейных уравнений"):

$$T^1 = \frac{4n^3 + 7n^2 - n - 22}{6}\tau + \frac{8 \cdot \left(2n^3 + 53n^2 + 49n - 110\right)}{6\beta} \qquad (3.5)$$

где τ есть время выполнения одной базовой вычислительной операции, а β есть пропускная способность канала доступа к оперативной памяти.

Заполним таблицу сравнения реального времени выполнения со временем, которое может быть получено по формуле (3.5). Для определения времени выполнения одной операции τ, как и при выполнении предыдущих лабораторных работ, выберем эксперимент, в котором матрица A и вектор b полностью умещаются в кэш. Далее время выполнения этого эксперимента поделим на число выполненных операций (первое слагаемое в формуле (3.5)). Таким образом, вычислим время выполнения одной операции. Оценка для величины пропускной способности оперативной памяти была получена при выполнении лабораторной работы 1 (данная величина не зависит от типа алгоритма). Далее, используя найденные значения параметров τ и β, можно вычислить теоретическое время выполнения для любых экспериментов.

Удалено:

Вычислите теоретическое время выполнения алгоритма Гаусса. Результаты занесите в таблицу:

Таблица 3.2. Сравнение реального времени выполнения последовательного алгоритма Гаусса со временем, вычисленным теоретически

Отформатировано: интервал После: 6 пт

Время выполнения одной операции τ (сек):			
Номер теста	Размер матрицы	Время работы (сек)	Теоретическое время (сек)
1	10		
2	100		
3	500		
4	1000		
5	1500		
6	2000		
7	2500		
8	3000		

Упражнение 4 – Разработка параллельного алгоритма Гаусса

Определение подзадач

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В этом случае, в качестве *базовой подзадачи* можно принять все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b.

Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить *(n-1)* итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду.

Выполнение итерации i, $0 \le i < n-1$, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Поскольку строки матрицы A распределены по подзадачам, для поиска максимального значения подзадачи с номерами k, $k > i$, должны обменяться своими элементами при исключаемой переменной x_i. После сбора всех необходимых данных в каждой подзадаче может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Зная ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении **обратного хода** метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, $0 \le i < n-1$, определяет значение своей переменной x_i, это значение должно быть использовано всеми подзадачами с номерами k, $k < 1$, для выполнения корректировки значений элементов вектора b.

Удалено:

Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и сбалансированным объемом передаваемых данных. В случае, когда размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных вычислительных элементов (т.е., $p < n$), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы.

Упражнение 5 - Реализация параллельного алгоритма Гаусса решения систем линейных уравнений

При выполнении этого упражнения Вам будет предложено разработать параллельный алгоритм Гаусса для решения систем линейных уравнений. При работе с этим упражнением Вы

- Получите опыт разработки параллельных программ,
- Познакомитесь с новыми языковыми конструкциями OpenMP.

Удалено: сложных

Задание 1 – Открытие проекта ParallelGauss

Откройте проект **ParallelGauss**, последовательно выполняя следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2005**, если оно еще не запущено,
- В меню File выполните команду **Open→Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **c:\MsLabs\ParallelGauss**,
- Дважды щелкните на файле **ParallelGauss.sln** или подсветите его выполните команду **Open**.

После того, как Вы открыли проект, в окне Solution Explorer (Ctrl+Alt+L) дважды щелкните на файле исходного кода **ParallelGauss.cpp**, как это показано на рисунке 3.8. После этих действий код, который вам предстоит модифицировать, будет открыт в рабочей области Visual Studio.

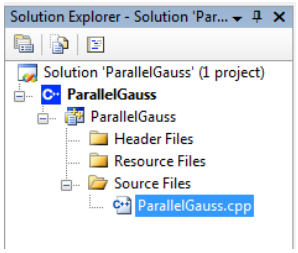

Рис. 3.8. Открытие файла **ParallelGauss.cpp** с использованием Solution Explorer

В файле **ParallelGauss.cpp** подключаются необходимые библиотеки, также в этом файле расположена главная функция (*main*) будущего параллельного приложения, которая содержит строки объявления необходимых переменных:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  printf("Parallel Gauss algorithm for solving linear systems\n");

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Также в файле **ParallelGauss.cpp** расположены функции, перенесенные сюда из проекта, содержащего последовательный алгоритм Гаусса: *DummyDataInitialization*, *RandomDataInitialization*, *SerialResultCalculation*, *SerialGaussianElimination*, *SerialBackSubstitution*, *SerialEliminateRows*, *FindPivotRow*. Первые две из этих функций будут использоваться в параллельном приложении для инициализации исходных объектов. Остальные функции будут служить основой при разработке параллельного алгоритма, так как при использовании технологии OpenMP получение параллельных вариантов программы во многих случаях осуществляется в результате добавления директив OpenMP в исходный последовательный код.

В данном параллельном приложении для печати матриц и векторов, как и ранее, будем пользоваться функциями *PrintMatrix* и *PrintVector*, реализация этих функций также перенесена в данное параллельное приложение. Кроме того, помещены заготовки для функций инициализации процесса вычислений (*ProcessInitialization*) и завершения процесса (*ProcessTermination*).

Скомпилируйте и запустите приложение стандартными средствами Visual Studio. Убедитесь в том, что в командную консоль выводится приветствие: "`Parallel Gauss algorithm for solving linear systems`".

Задание 2 – Подготовка функции реализации метода Гаусса

Прямой ход параллельного метода Гаусса, как и последовательный, путем эквивалентных преобразований приводит матрицу системы линейных уравнений к верхнему треугольному виду. На каждой итерации выполняемых преобразований для выбора ведущей строки применяют *метод главных элементов* (см. упражнение 2), в соответствии с которым в качестве ведущей выбирается строка, содержащая максимальный по абсолютному значению элемент очередного столбца матрицы.

Для запоминания порядка выбора ведущих строк в параллельном алгоритме, как и в последовательном, введем массив *pPivotPos*, в *i*-ом элементе которого будем хранить номер строки, которая была выбрана в качестве ведущей при выполнении *i*-ой итерации прямого хода алгоритма Гаусса. Также определим массив – *pPivotIter*, в каждом элементе которого *pPivotIter*[*i*] будем хранить номер итерации, на которой строка с номером *i* выбиралась в качестве ведущей. Изначально массив *pPivotIter* заполним элементами, равными -1 (т.е. значение -1 в элементе массива *pPivotIter*[*i*] означает, что строка с номером *i* еще не выбиралась в качестве ведущей).

Объявим соответствующие массивы как глобальные, выделим память для этих массивов, а затем освободим выделенную память. Предполагается, что между выделением и удалением массива производятся прямой и обратный ход метода Гаусса:

```
int* pPivotPos;  // The number of pivot rows selected at the iterations
int* pPivotIter; // The iterations, at which the rows were pivots

// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {
```


Отформатировано: интервал После: 6 пт

Примечание [Г2]: Сделать комментарии такими же как в последовательной программе

Отформатировано: английский (США)

Отформатировано: английский (США)

Примечание [Г3]: Смм. Пред. замечание

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: интервал Перед: 6 пт

Удалено: ,

Удалено: е

Удалено: й

Отформатировано: русский

Удалено: предполагает, что из

Удалено: ого

Удалено: а можно сделать параллельный путем внесения в код прагм.

Отформатировано: Шрифт: не курсив

Отформатировано: Шрифт: не курсив

Отформатировано: Шрифт: не курсив

Отформатировано: Шрифт: не курсив

Отформатировано: интервал После: 6 пт

Удалено: ¶ //

Отформатировано: русский

Отформатировано: французский (Франция)

```
  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pPivotIter[i] = -1;
  }

  // Gaussian elimination
  // Back substitution

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}
```

Примечание [ГВП4]: Где прямой ход

Примечание [E5R4]: На данной части разработки вызов дополнительных функций не требуется в дальнейшем появится.

Добавьте вызов функции *ParallelResultCalculation* в функцию *main*, как показано дальше:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  ParallelResultCalculation(pMatrix, pVector, pResult, Size);

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Задание 3 – Определение “локальных” для потоков ведущих строк

Напомним, что в прямом ходе метода Гаусса на каждой итерации алгоритма осуществляется поиск ведущей строки. Рассмотрим процедуру определения ведущей строки. На каждой *i*-ой итерации прямого хода метода Гаусса необходимо просмотреть строки, подлежащие обработке, и выбрать среди них строку, у которой в *i*-ом столбце расположен максимальный по абсолютному значению элемент. Каждый поток параллельной программы отвечает за вычисления, производимые над совокупностью строк матрицы линейной системы и соответствующими элементами векторов. Необходимо изменить код таким образом, чтобы на первом этапе все потоки параллельно определили «локальную» ведущую строку в рамках своей полосы системы.

Для этого объявите новую структуру для хранения «локальной» ведущей строки *TThreadPivotRow*. Полями этой структуры являются номер «локальной» ведущей строки в матрице линейной системы и значение, расположенное в столбце с исключаемой неизвестной.

```
typedef struct
{
  int PivotRow;
  double MaxValue;
} TThreadPivotRow;
```

Для того чтобы выбор ведущей строки выполнялся потоками параллельно, в функции поиска ведущего элемента объявите параллельную секцию при помощи директивы **parallel**. Внутри параллельной секции объявите переменную типа *TThreadPivotRow* для хранения «локальной» ведущей строки. Поскольку данная переменная объявлена внутри параллельной секции, локальные копии этой переменной будут созданы во всех потоках. Поле *PivotRow* объявленной переменной *ThreadPivotRow* проинициализируем значением равным -1.

Примечание [ГВП6]: См. раздел про OpenMP - стиль

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

```
// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1;    // The index of the pivot row
```

```
  double MaxValue =  0; // The value of the pivot element
  int i;                // Loop variable

  // Choose the row, that stores the maximum element
#pragma omp parallel
  {
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
  } // pragma omp parallel
  return PivotRow;
}
```

Далее необходимо написать параллельный цикл поиска “локальной” ведущей строки. Для распределения итераций цикла выбора ведущей строки между потоками можно воспользоваться директивой **for**.

```
// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1;    // The index of the pivot row
  double MaxValue =  0; // The value of the pivot element
  int i;                // Loop variable

  // Choose the row, that stores the maximum element
#pragma omp parallel
  {
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
#pragma omp for
    for (i=0; i<Size; i++) {
      if ((pSerialPivotIter[i] == -1) &&
        (fabs(pMatrix[i*Size+Iter]) > ThreadPivotRow.MaxValue)) {
        ThreadPivotRow.PivotRow = i;
        ThreadPivotRow.MaxValue = fabs(pMatrix[i*Size+Iter]);
      }
    }
  } // pragma omp parallel
  return PivotRow;
}
```

Для того чтобы убедится в правильности выбора ведущего элемента каждый поток должен вывести свой идентификатор и результат своего поиска. Возможный окончательный для этого этапа вид функции представлен ниже:

```
// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1;    // The index of the pivot row
  double MaxValue =  0; // The value of the pivot element
  int i;                // Loop variable

  // Choose the row, that stores the maximum element
#pragma omp parallel
  {
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
#pragma omp for
    for (i=0; i<Size; i++) {
      if ((pSerialPivotIter[i] == -1) &&
        (fabs(pMatrix[i*Size+Iter]) > ThreadPivotRow.MaxValue)) {
        ThreadPivotRow.PivotRow = i;
        ThreadPivotRow.MaxValue = fabs(pMatrix[i*Size+Iter]);
      }
    }
```

```
    printf("Local thread (id = %i) pivot row : %i\n",
    omp_get_thread_num(), ThreadPivotRow.PivotRow);
  } // pragma omp parallel
  return PivotRow;
}
```

Примечание [I7]: Нужно печатать еще и номер потока

В функции *main* для тестирования функции добавьте печать матрицы и вектора:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  ParallelResultCalculation(pMatrix, pVector, pResult, Size);

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

В функции *ParallelResultCalculation*, добавьте вызов функции определения ведущей строки как показано ниже:

```
// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }

  ParallelFindPivotRow(pMatrix, Size, 0);

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}
```

Отформатировано: французский (Франция)

Откомпилируйте и запустите программу. Один из возможных результатов выполнения программы представлен ниже (результат может меняться в зависимости от количества вычислительных элементов в системе и способа инициализации генератора случайных чисел):

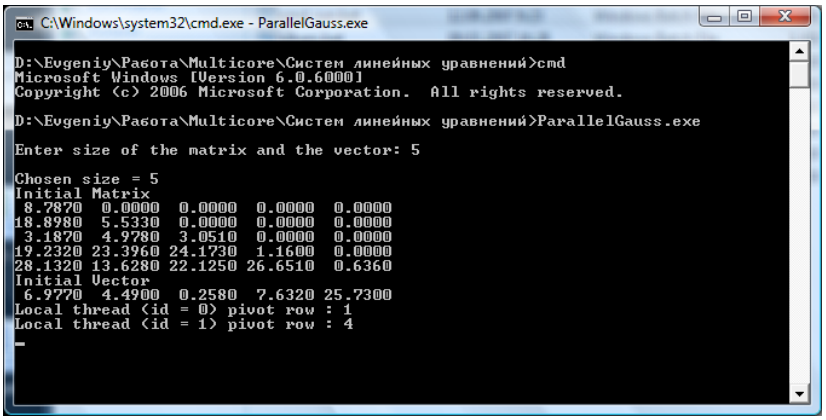

Рис. 3.9. Результат поиска “локальных” ведущих строк при использовании двух потоков

Задание 3 – Определение “глобальной” ведущей строки

После того, как все потоки определили «локальную» ведущую строку, до завершения параллельной секции необходимо выполнить редукцию полученных значений и выбрать «глобальную» ведущую строку. Для этого необходимо просмотреть выбранные «локальные» ведущие строки и выбрать среди них строку с максимальным значением поля *MaxValue*. Выбор глобальной ведущей строки можно обеспечить при помощи механизма критических секций (директива **critical**). Код, расположенный внутри критической секции, в каждый момент времени выполняется только одним потоком. После того, как один из потоков закончит выполнение критической секции, к ее выполнению может приступить другой поток. В конечном итоге, критическая секция будет выполнена всеми потоками.

Один из вариантов использования критических секций для реализации редукции «локальных» ведущих строк представлен ниже:

```
// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1;    // The index of the pivot row
  double MaxValue =  0; // The value of the pivot element
  int i;                // Loop variable

  // Choose the row, that stores the maximum element
#pragma omp parallel
  {
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
#pragma omp for
    for (i=0; i<Size; i++) {
      if ((pPivotIter[i] == -1) &&
        (fabs(pMatrix[i*Size+Iter]) > ThreadPivotRow.MaxValue)) {
        ThreadPivotRow.PivotRow = i;
        ThreadPivotRow.MaxValue = fabs(pMatrix[i*Size+Iter]);
      }
    }
    printf("Local thread (id = %i) pivot row : %i\n",
      omp_get_thread_num(), ThreadPivotRow.PivotRow);

#pragma omp critical
    {
      if (ThreadPivotRow.MaxValue > MaxValue){
        MaxValue = ThreadPivotRow.MaxValue;
        PivotRow = ThreadPivotRow.PivotRow;
      }
    } // pragma omp critical
  } // pragma omp parallel
```

Примечание [I8]: Нужно печатать еще и номер потока

```
  return PivotRow;
}
```

Для контроля правильности выполнения параллельного алгоритма поиска ведущей строки добавим печать номера найденной строки:

```
// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }

  int GlobalPivotPos;
  GlobalPivotPos = ParallelFindPivotRow(pMatrix, Size, 0);
  printf("Global pivot row : %i\n", GlobalPivotPos);

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}
```

Откомпилируйте и запустите приложение. Один из возможных результатов выполнения программы представлен ниже (результат может меняться в зависимости от количества вычислительных элементов в системе и способа инициализации генератора случайных чисел):

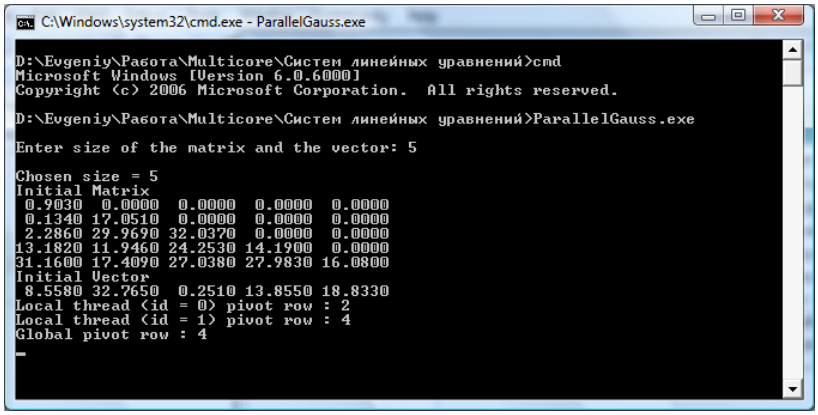

Рис. 3.10. Результат поиска “глобальной” ведущей строки

Задание 4 – Параллельное исключение очередной неизвестной системы линейных уравнений

Выбранная «глобальная» ведущая строка используется далее для исключения неизвестной во всех строках, подлежащих обработке. Реализуем параллельный вариант данной части алгоритма в функции *ParallelEliminateColumns*.

За обработку одной строки матрицы линейной системы отвечает одна итерация внешнего цикла по переменной *i*. Поскольку преобразования строк осуществляются независимо, то эти вычисления можно распределить между параллельными потоками согласно вычислительной схеме параллельного алгоритма. Для этого воспользуемся директивой **parallel for** для распределения итераций внешнего цикла между потоками. При этом переменная *PivotFactor*, которая используется для хранения множителя, на который умножается текущая строка, должна быть локализована (параметр *private* директивы **parallel for**).

Один из возможных вариантов параллельной реализации функции приведен ниже:

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: французский (Франция)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Примечание [I9]: Опять же, пример с единицами не показателен

```
// Column elimination
void ParallelColumnElimination (double* pMatrix, double* pVector,
int Pivot, int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private (PivotFactor)
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}
```

Итерации цикла распределяются между потоками поровну. Так, например, если для выполнения цикла, содержащего 100 итераций, используется два потока, то первый поток отвечает за выполнение итераций с номерами от 0 до 49, а второй – за выполнение итераций с номерами от 50 до 99. В случае, если итерации цикла неравнозначны, такое распределение может приводить к неравномерному распределению вычислительной нагрузки.

При выполнении прямого хода метода Гаусса после выбора ведущей строки вычисления производятся только над строками, подлежащими обработке (строка с номером *i* подлежит обработке, если значение элемента массива *pSerialPivotIter[i]* равно -1, что означает что данная строка не выбиралась ранее в качестве ведущей). Следовательно, итерации цикла обладают разной вычислительной сложностью.

Для того, чтобы гарантировать равномерную загрузку вычислительных элементов, можно явно указать, как именно должны распределяться итерации между потоками при помощи параметра *schedule* директивы **parallel for**:

```
// Column elimination
void ParallelColumnElimination (double* pMatrix, double* pVector,
int Pivot, int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private(PivotFactor) schedule(dynamic,1)
  for (int i=0; i<Size; i++) {
    …
  }
}
```

В данном случае организуется подобие «очереди итераций»: каждый поток берет на выполнение текущую итерацию из очереди, как только итерация выполнена, происходит новое обращение к очереди за новой итерацией. Такой способ распределения вычислительной нагрузки гарантирует балансировку вычислительных элементов.

Для демонстрации работы алгоритма измените функцию поиска решения системы линейных уравнений следующем образом:

```
// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }

  int GlobalPivotPos;
  GlobalPivotPos = ParallelFindPivotRow(pMatrix, Size, 0);
  printf("Global pivot row : %i\n", GlobalPivotPos);
```

```
  pPivotPos[0] = GlobalPivotPos;
  pPivotIter[GlobalPivotPos] = 0;
  ParallelColumnElimination (pMatrix, pVector, GlobalPivotPos, 0, Size);

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}
```

Кроме того, добавьте в функцию *main* печать матрицы после вызова функции поиска решения методом Гаусса:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  ParallelResultCalculation(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  printf ("Recalculated Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Recalculated Vector \n");
  PrintVector(pVector, Size);

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Откомпилируйте и запустите программу. Один из возможных результатов выполнения программы представлен ниже (результат может меняться в зависимости от количества вычислительных элементов в системе и способа инициализации генератора случайных чисел):

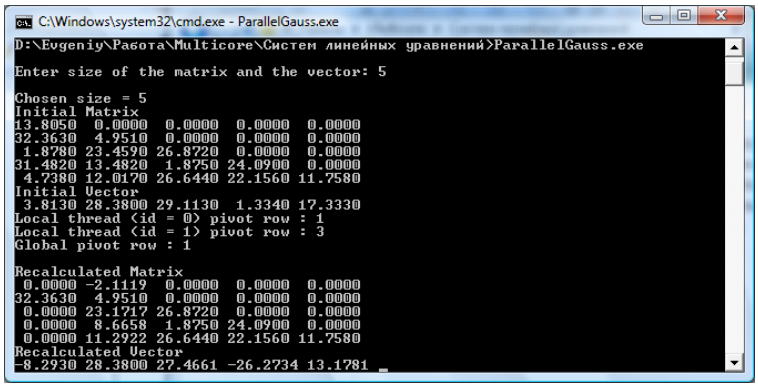

Рис. 3.11. Результат вычислений после выполнения первой итерации прямого хода метода Гаусса

Отформатировано: португальский (Бразилия)

Отформатировано: португальский (Бразилия)

Примечание [I10]: Странно это как-то.. получилось, что мы запустили одну итерацию метода Гаусса…

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: английский (США)

Примечание [E11R10]: Мы показали, что обнуление одной строки работает корректно

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: русский

Отформатировано: английский (США)

Отформатировано: английский (США)

Отформатировано: русский

Примечание [I12]: Как и раньше, пример с единицами не показателен

Задание 5 – Реализация прямого хода метода Гаусса

Функция, которая реализует параллельный вариант прямого хода метода Гаусса, практически не отличается от последовательного варианта. Отличие заключается в том, что вызываются параллельные варианты функций поиска ведущей строки (*ParallelFindPivotRow*) и исключения текущей неизвестной (*ParallelColumnElimination*).

Функция, выполняющая параллельный вариант прямого хода метода Гаусса, представлена ниже:

```
// Gaussian elimination
void ParallelGaussianElimination(double* pMatrix,double* pVector,int Size)
{
  int Iter;       // The number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter);
    pPivotPos[Iter] = PivotRow;
    pPivotIter[PivotRow] = Iter;
    ParallelColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
}
```

Для демонстрации работы прямого хода метода Гауса измените функцию *ParallelResultCalculation* следующим образом:

```
// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }

  ParallelGaussianElimination(pMatrix, pVector, Size);

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}
```

Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский

Откомпилируйте и запустите программу. В результате работы функции должны получиться верхне-треугольные матрицы с учетом надлежащего переупорядочивания строк. Один из возможных результатов выполнения программы представлен ниже (результат может меняться в зависимости от способа инициализации генератора случайных чисел):

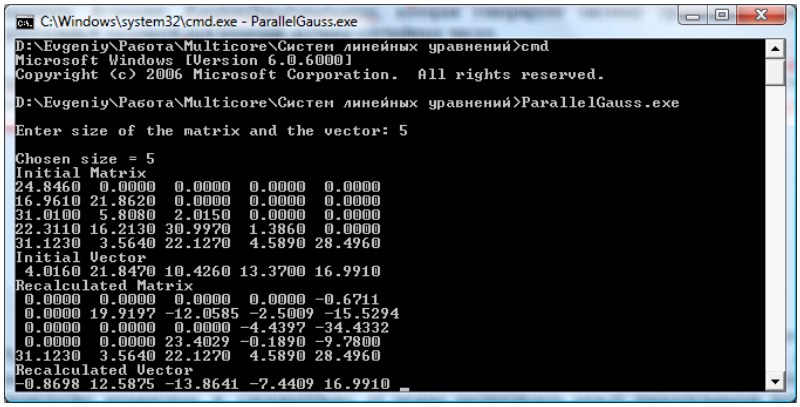

Рис. 3.12. Результат работы прямого хода метода Гаусса при случайной генерации коэффициентов матрицы и результирующего вектора (матрица станет верхне-треугольной, если поставить строки в порядке (5, 2, 4, 3, 1))

Задание 6 – Реализация обратного хода метода Гаусса

После реализации прямого хода метода Гаусса необходимо разработать функцию обратного хода *ParallelBackSubstitution*. Итерации обратного хода метода Гаусса должны выполняться строго последовательно. Однако обработка строк линейной системы на каждой итерации обратного хода происходит независимо, и, следовательно, ее можно распределить между параллельными потоками согласно вычислительной схеме параллельного алгоритма.

При помощи директивы **parallel for** распределите между потоками параллельной программы итерации внутреннего цикла, при этом переменная *Row*, которая хранит номер текущей обрабатываемой строки, должна быть локализована. Код функции представлен ниже:

```
// Back substution
void ParallelBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
#pragma omp parallel for private (Row)
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[Row] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}
```

Для завершения разработки метода Гаусса добавьте вызов функции *ParallelBackSubstitution*, после вызова функции прямого хода метода Гаусса:

```
// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pPivotIter[i] = -1;
  }

  ParallelGaussianElimination(pMatrix, pVector, Size);
  ParallelBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}
```

В функции *main* добавьте вывод решения системы линейных уравнений:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);
```

Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: русский
Отформатировано: английский (США)
Отформатировано: французский (Франция)
Отформатировано: английский (США)
Отформатировано: английский (США)

```
  // The matrix and the vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  ParallelResultCalculation(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  printf ("Recalculated Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Recalculated Vector \n");
  PrintVector(pVector, Size);
  printf("\nResult Vector \n");
  PrintVector(pResult, Size);

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Отформатировано: английский (США)

Отформатировано: английский (США)

Откомпилируйте и запустите программу. Один из возможных результатов выполнения программы представлен ниже (результат может меняться в зависимости от способа инициализации генератора случайных чисел):

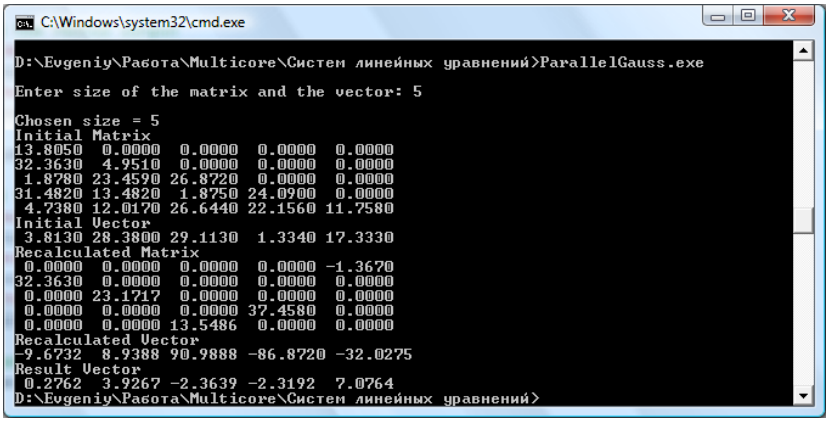

Рис. 3.13. Результат работы параллельного метода Гаусса решения системы линейных уравнений

Задание 7 – Проверка правильности работы программы

Теперь, после реализации параллельного метода Гаусса, необходимо проверить правильность выполнения алгоритма. Для этого разработаем функцию *TestResult*. Для проверки правильности выполнения алгоритма необходимо умножить матрицу линейной системы на полученный вектор неизвестных (с учетом порядка элементов в нем), а затем поэлементно сравнить вектор, полученный в результате такого умножения с заданным вектором правых частей *pVector*. Получение каждого элемента результирующего вектора требует выполнения последовательности умножений и сложений дробных чисел. Порядок выполнения этих действий может повлиять на величину машинной погрешности. Поэтому в данном случае нельзя проверять элементы двух векторов (*pRightPartVector* и *pVector*) на равенство. Введем допустимую величину расхождения результатов *Accuracy*. Вектора будем считать равными в том случае, когда соответствующие элементы отличаются не более чем на величину допустимой погрешности *Accuracy*.

Функция *TestResult* должна иметь доступ к матрице системы линейных уравнений *pMatrix* и вектору правых частей *pVector* и, следовательно, иметь вид:

```
// Function for testing the result
```

```
void TestResult (double* pMatrix, double* pVector, double* pResult,
  int Size) {
  /* Buffer for storing the vector, that is a result of multiplication
     of the linear system matrix by the vector of unknowns */
  double* pRightPartVector;
  // Flag, that shows wheather the right parts vectors are identical or not
  int equal = 0;
  double Accuracy = 1.e-6; // Comparison accuracy

  pRightPartVector = new double [Size];
  for (int i=0; i<Size; i++) {
    pRightPartVector[i] = 0;
    for (int j=0; j<Size; j++) {
      pRightPartVector[i] +=
        pMatrix[i*Size+j]*pResult[j];
    }
  }

  for (int i=0; i<Size; i++) {
    if (fabs(pRightPartVector[i]-pVector[i]) > Accuracy))
      equal = 1;
  }
  if (equal == 1)
    printf("The result of the parallel Gauss algorithm is NOT correct."
           "Check your code.");
  else
    printf("The result of the parallel Gauss algorithm is correct.");
  delete [] pRightPartVector;
}
```

Результатом работы этой функции является печать диагностического сообщения. Используя эту функцию, можно проверять результат работы параллельного алгоритма независимо от того, насколько велики исходные объекты при любых значениях исходных данных.

Закомментируйте вызовы функций, использующих отладочную печать, которые ранее использовались для контроля правильности выполнения этапов параллельного приложения. Скомпилируйте и запустите приложение. Задавайте различные объемы исходных данных. Убедитесь в том, что приложение работает правильно.

Задание 8 – Проведение вычислительных экспериментов

Определим время выполнения параллельного алгоритма. Для этого добавим в программный код замеры времени, воспользуемся стандартной функцией таймированяи OpenMP, которая возвращает время в секундах, прошедшее от определенного момента в прошлом:

```
double omp_get_wtime();
```

Измените функцию *main* следующим образом:

```
void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  Start = omp_get_wtime();
  ParallelResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = omp_get_wtime();
  Duration = Finish-Start;

  // Testing the result
  TestResult(pMatrix, pVector, pResult, Size);
```

Отформатировано: английский (США)

```
  // Printing the time spent by parallel Gauss algorithm
  printf("\n Time of execution: %f\n", Duration);

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Отформатировано: английский (США)

Отформатировано: английский (США)

Очевидно, что таким образом будет распечатано то время, которое было затрачено на выполнение вычислений.

Добавьте выделенный фрагмент кода в тело основной функции приложения. Скомпилируйте и запустите приложение. Заполните таблицу:

Таблица 3.3. Время выполнения параллельного алгоритма Гаусса решения систем линейных уравнений и ускорение

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм			
			2 ядра		4 ядра	
			Время	Ускорение	Время	Ускорение
1	10					
2	100					
3	500					
4	1000					
5	1500					
6	2000					
7	2500					
8	3000					

В графу "Последовательный алгоритм" внесите время выполнения последовательного алгоритма, полученное при вполнении задания 7 упражнении 3. Для того, чтобы вычислить ускорение, разделите время выполнения последовательного алгоритма на время выполнения параллельного алгоритма. Результат поместите в соответствующую графу таблицы.

Для того, чтобы оценить теоретическое время выполнения параллельного алгоритма, реализованного согласно вычислительной схеме, приведенной в упражнении 4, можно воспользоваться следующим соотношением:

$$T^p = \frac{4n^3 + 7n^2 - n - 22}{6p}\tau + 3p\tau + \frac{8\cdot\left(2n^3 + 53n^2 + 49n - 110\right)}{6\beta} + 3(n-1)\delta \qquad (3.6)$$

(подробный вывод этой формулы приведен в разделе 9 "Параллельные методы решения систем линейных уравнений" учебных материалов курса). Здесь n – размер системы линейных уравнений, p – количество вычислительных элементов, τ – время выполнения одной скалярной операции (значение было нами вычислено при тестировании последовательного алгоритма), β – пропускная способность канала доступа к оперативной памяти, δ – время, необходимое на организацию и закрытие параллельной секции (методика оценки последнего параметра описана в разделе 7 учебных материалов курса).

Вычислите теоретическое время выполнения параллельного алгоритма по формуле (3.6). Результаты занесите в таблицу 3.4:

Таблица 3.4. Сравнение реального времени выполнения параллельного алгоритма со временем, вычисленным теоретически

Номер теста	Порядок системы	2 ядра		4 ядра	
		Модель	Эксперимент	Модель	Эксперимент
1	10				
2	100				
3	500				
4	1000				
5	1500				
6	2000				
7	2500				
8	3000				

Контрольные вопросы

- Насколько сильно отличаются время, затраченное на выполнение последовательного и параллельного алгоритма? Почему?
- Получилось ли ускорение в случае, когда порядок системы уравнений равен 10? Почему?
- Насколько хорошо совпадают время, полученное теоретически, и реальное время выполнения алгоритма? В чем может состоять причина несовпадений?

Задания для самостоятельной работы

1. Изучите метод сопряженных градиентов решения систем линейных уравнений. Выполните реализацию последовательного и параллельного вариантов этого метода.

Приложение 1. Программный код последовательного алгоритма Гаусса решения линейных систем

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <windows.h>

// function that LongInt convert to double
double LiToDouble (LARGE_INTEGER x) {
  double    result    =    ((double)x.HighPart)    *    4.294967296E9    +
(double)((x).LowPart);
  return result;
}

// function that to note the time
double GetTime() {
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency (&lpFrequency);
  QueryPerformanceCounter (&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

int*  pSerialPivotPos;    //  The  Number  of  pivot  rows  selected  at  the
iterations
int* pSerialPivotIter; // The Iterations, at which the rows were pivots

// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++) {
    pVector[i] = i+1;
      for (j=0; j<Size; j++) {
      if (j <= i)
          pMatrix[i*Size+j] = 1;
        else
          pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for random initialization of the matrix and the vector elements
void RandomDataInitialization (double* pMatrix, double* pVector, int Size)
{
  int i, j;  // Loop variables
  srand(unsigned(clock()));
```

```
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
      for (j=0; j<Size; j++) {
        if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
        else
        pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for memory allocation and definition of the objects elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
  // Setting the size of the matrix and the vector
  do {
    printf("\nEnter size of the matrix and the vector: ");
    scanf("%d", &Size);
    printf("\nChosen size = %d \n", Size);

      if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];

  // Initialization of the matrix and the vector elements
  DummyDataInitialization(pMatrix, pVector, Size);
  //RandomDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
        printf("%7.4f ", pMatrix[i*RowCount+j]);
      printf("\n");
  }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1; // The index of the pivot row
  int MaxValue =  0; // The value of the pivot element
  int i;             // Loop variable

  // Choose the row, that stores the maximum element
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
       (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
```

Отформатировано: французский (Франция)

```
    }
  }
  return PivotRow;
}

// Column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}

// Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
}

// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}



// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pSerialPivotPos  = new int [Size];
  pSerialPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }
  // Gaussian elimination
```

```
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pSerialPivotPos;
  delete [] pSerialPivotIter;
}

// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult)
{
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}


void main() {
  double* pMatrix;  // The matrix of the linear system
  double* pVector;  // The right parts of the linear system
  double* pResult;  // The result vector
  int Size;         // The sizes of the initial matrix and the vector
  double start, finish;
  double duration;

  printf("Serial Gauss algorithm for solving linear systems\n");
  // Memory allocation and definition of objects' elements
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  // The matrix and the vector output
  printf ("Initial Matrix \n");
  PrintMatrix(pMatrix, Size, Size);
  printf("Initial Vector \n");
  PrintVector(pVector, Size);

  // Execution of Gauss algorithm
  start = omp_get_wtime();
  SerialResultCalculation(pMatrix, pVector, pResult, Size);
  finish = omp_get_wtime();
  duration = finish-start;

  // Printing the result vector
  printf ("\n Result Vector: \n");
  PrintVector(pResult, Size);

  // Printing the execution time of Gauss method
  printf("\n Time of execution: %f\n", duration);

  // Computational process termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Приложение 2. Программный код параллельного алгоритма Гаусса решения линейных систем

```
#include "omp.h"
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
```

```
#include <windows.h>

// function that LongInt convert to double
double LiToDouble (LARGE_INTEGER x) {
  double    result    =    ((double)x.HighPart)    *    4.294967296E9    +
(double)((x).LowPart);
  return result;
}

// function that to note the time
double GetTime() {
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency (&lpFrequency);
  QueryPerformanceCounter (&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

int*  pSerialPivotPos;    //  The  Number  of  pivot  rows  selected  at  the
iterations
int* pSerialPivotIter; // The Iterations, at which the rows were pivots
int* pPivotPos;  // The Number of pivot rows selected at the iterations
int* pPivotIter; // The Iterations, at which the rows were pivots


// Function for simple initialization of the matrix and the vector elements
void DummyDataInitialization (double* pMatrix, double* pVector, int Size) {
  int i, j;  // Loop variables

  for (i=0; i<Size; i++) {
    pVector[i] = i+1;
      for (j=0; j<Size; j++) {
      if (j <= i)
          pMatrix[i*Size+j] = 1;
        else
          pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for random initialization of the matrix and the vector elements
void RandomDataInitialization (double* pMatrix, double* pVector, int Size)
{
  int i, j;  // Loop variables
  srand(unsigned(clock()));
  for (i=0; i<Size; i++) {
    pVector[i] = rand()/double(1000);
      for (j=0; j<Size; j++) {
        if (j <= i)
        pMatrix[i*Size+j] = rand()/double(1000);
        else
        pMatrix[i*Size+j] = 0;
      }
  }
}

// Function for memory allocation and definition of the objects elements
void ProcessInitialization (double* &pMatrix, double* &pVector,
    double* &pResult, int &Size) {
  // Setting the size of the matrix and the vector
  do {
    printf("\nEnter size of the matrix and the vector: ");
    scanf("%d", &Size);
    printf("\nChosen size = %d \n", Size);
```

Отформатировано: французский (Франция)

```
      if (Size <= 0)
      printf("\nSize of objects must be greater than 0!\n");
  } while (Size <= 0);

  // Memory allocation
  pMatrix = new double [Size*Size];
  pVector = new double [Size];
  pResult = new double [Size];

  // Initialization of the matrix and the vector elements
  //DummyDataInitialization(pMatrix, pVector, Size);
  RandomDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix (double* pMatrix, int RowCount, int ColCount) {
  int i, j; // Loop variables
  for (i=0; i<RowCount; i++) {
    for (j=0; j<ColCount; j++)
        printf("%7.4f ", pMatrix[i*RowCount+j]);
      printf("\n");
  }
}

// Function for formatted vector output
void PrintVector (double* pVector, int Size) {
  int i;
  for (i=0; i<Size; i++)
    printf("%7.4f ", pVector[i]);
}

// Finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1; // The index of the pivot row
  int MaxValue =  0; // The value of the pivot element
  int i;             // Loop variable

  // Choose the row, that stores the maximum element
  for (i=0; i<Size; i++) {
    if ((pSerialPivotIter[i] == -1) &&
       (fabs(pMatrix[i*Size+Iter]) > MaxValue)) {
      PivotRow = i;
      MaxValue = fabs(pMatrix[i*Size+Iter]);
    }
  }
  return PivotRow;
}

// Column elimination
void SerialColumnElimination (double* pMatrix, double* pVector, int Pivot,
  int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
  for (int i=0; i<Size; i++) {
    if (pSerialPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
```

```
}

// Gaussian elimination
void SerialGaussianElimination(double* pMatrix,double* pVector,int Size) {
  int Iter;       // The number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = FindPivotRow(pMatrix, Size, Iter);
    pSerialPivotPos[Iter] = PivotRow;
    pSerialPivotIter[PivotRow] = Iter;
    SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
}

// Back substution
void SerialBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pSerialPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
    for (int j=0; j<i; j++) {
      Row = pSerialPivotPos[j];
      pVector[j] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}



// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pSerialPivotPos  = new int [Size];
  pSerialPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
  }
  // Gaussian elimination
  SerialGaussianElimination (pMatrix, pVector, Size);
  // Back substitution
  SerialBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pSerialPivotPos;
  delete [] pSerialPivotIter;
}


// Function for computational process termination
void ProcessTermination (double* pMatrix, double* pVector, double* pResult)
{
  delete [] pMatrix;
  delete [] pVector;
  delete [] pResult;
}
typedef struct
{
```

```
  int PivotRow;
  double MaxValue;
} TThreadPivotRow;

// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
  int PivotRow = -1;    // The index of the pivot row
  double MaxValue =  0; // The value of the pivot element
  int i;                // Loop variable

  // Choose the row, that stores the maximum element
#pragma omp parallel
  {
    TThreadPivotRow ThreadPivotRow;
    ThreadPivotRow.MaxValue = 0;
    ThreadPivotRow.PivotRow = -1;
#pragma omp for
    for (i=0; i<Size; i++) {
      if ((pPivotIter[i] == -1) &&
        (fabs(pMatrix[i*Size+Iter]) > ThreadPivotRow.MaxValue)) {
        ThreadPivotRow.PivotRow = i;
        ThreadPivotRow.MaxValue = fabs(pMatrix[i*Size+Iter]);
      }
    }
//    printf("Local thread pivot row : %i\n", ThreadPivotRow.PivotRow);
      #pragma omp critical
  {
    if (ThreadPivotRow.MaxValue > MaxValue){
      MaxValue = ThreadPivotRow.MaxValue;
      PivotRow = ThreadPivotRow.PivotRow;
    }
  } // pragma omp critical

  } // pragma omp parallel
  return PivotRow;
}

// Column elimination
void ParallelColumnElimination (double* pMatrix, double* pVector,
int Pivot, int Iter, int Size) {
  double PivotValue, PivotFactor;
  PivotValue = pMatrix[Pivot*Size+Iter];
#pragma omp parallel for private(PivotFactor) schedule(dynamic,1)
  for (int i=0; i<Size; i++) {
    if (pPivotIter[i] == -1) {
      PivotFactor = pMatrix[i*Size+Iter] / PivotValue;
      for (int j=Iter; j<Size; j++) {
        pMatrix[i*Size + j] -= PivotFactor * pMatrix[Pivot*Size+j];
      }
      pVector[i] -= PivotFactor * pVector[Pivot];
    }
  }
}

// Gaussian elimination
void  ParallelGaussianElimination(double*  pMatrix,double*  pVector,int  Size)
{
  int Iter;       // The number of the iteration of the gaussian
                  // elimination
  int PivotRow;   // The number of the current pivot row
  for (Iter=0; Iter<Size; Iter++) {
    // Finding the pivot row
    PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter);
```

```
    pPivotPos[Iter] = PivotRow;
    pPivotIter[PivotRow] = Iter;
    ParallelColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
  }
}

// Back substution
void ParallelBackSubstitution (double* pMatrix, double* pVector,
  double* pResult, int Size) {
  int RowIndex, Row;
  for (int i=Size-1; i>=0; i--) {
    RowIndex = pPivotPos[i];
    pResult[i] = pVector[RowIndex]/pMatrix[Size*RowIndex+i];
#pragma omp parallel for private (Row)
    for (int j=0; j<i; j++) {
      Row = pPivotPos[j];
      pVector[Row] -= pMatrix[Row*Size+i]*pResult[i];
      pMatrix[Row*Size+i] = 0;
    }
  }
}

// Function for testing the result
void TestResult (double* pMatrix, double* pVector, double* pResult,
  int Size) {
  /* Buffer for storing the vector, that is a result of multiplication
     of the linear system matrix by the vector of unknowns */
  double* pRightPartVector;
  // Flag, that shows wheather the right parts vectors are identical or not
  int equal = 0;
  double Accuracy = 1.e-6; // Comparison accuracy

  pRightPartVector = new double [Size];
  for (int i=0; i<Size; i++) {
    pRightPartVector[i] = 0;
    for (int j=0; j<Size; j++) {
      pRightPartVector[i] +=
      pMatrix[i*Size+j]*pResult[j];
    }
  }

  for (int i=0; i<Size; i++) {
        if (fabs(pRightPartVector[i]-pVector[i]) > Accuracy){
            equal = 1;
        }
  }
  if (equal == 1)
    printf("The result of the parallel Gauss algorithm is NOT correct."
           "Check your code.");
  else
    printf("The result of the parallel Gauss algorithm is correct.");
  delete [] pRightPartVector;
}

// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
  double* pResult, int Size) {

  // Memory allocation
  pPivotPos  = new int [Size];
  pPivotIter = new int [Size];
  for (int i=0; i<Size; i++) {
    pSerialPivotIter[i] = -1;
```

Отформатировано: французский (Франция)

```
  }

  ParallelGaussianElimination(pMatrix, pVector, Size);
  ParallelBackSubstitution (pMatrix, pVector, pResult, Size);

  // Memory deallocation
  delete [] pPivotPos;
  delete [] pPivotIter;
}

void main(int argc, char* argv[]) {
  double* pMatrix;        // The matrix of the linear system
  double* pVector;        // The right parts of the linear system
  double* pResult;        // The result vector
  int     Size;           // The size of the matrix and the vectors
  double Start, Finish, Duration;

  // Data initialization
  ProcessInitialization(pMatrix, pVector, pResult, Size);

  Start = omp_get_wtime();
  ParallelResultCalculation(pMatrix, pVector, pResult, Size);
  Finish = omp_get_wtime();
  Duration = Finish-Start;

  // Testing the result
  TestResult(pMatrix, pVector, pResult, Size);

  // Printing the time spent by parallel Gauss algorithm
  printf("\n Time of execution: %f\n", Duration);

  // Program termination
  ProcessTermination(pMatrix, pVector, pResult);
  getch();
}
```

Отформатировано: английский (США)

Стр. 13: [1] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [2] Отформатировано **Гергель** **05.10.2007 22:23:00**

интервал После: 6 пт

Стр. 13: [3] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [4] Отформатировано **Гергель** **05.10.2007 22:24:00**

интервал Перед: 6 пт, После: 6 пт

Стр. 13: [5] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [6] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [7] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [8] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [9] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [10] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [11] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [12] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [13] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [14] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [15] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 13: [16] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 24: [17] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [17] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [17] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [18] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [18] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

русский

Стр. 24: [18] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [18] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [18] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [19] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [19] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [19] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [19] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [19] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [19] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [20] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [20] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [20] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [21] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [21] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [21] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [21] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [21] Отформатировано Гергель 05.10.2007 22:08:00

русский

Стр. 24: [22] Отформатировано Гергель 05.10.2007 22:08:00

английский (США)

Стр. 24: [22] Отформатировано Гергель 05.10.2007 22:08:00

английский (США)

Стр. 24: [22] Отформатировано Гергель 05.10.2007 22:08:00

английский (США)

Стр. 24: [22] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 24: [22] Отформатировано **Гергель** **05.10.2007 22:08:00**

английский (США)

Стр. 24: [23] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [23] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [23] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [24] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [25] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [25] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [25] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [25] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [25] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [25] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [26] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [26] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [26] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

Стр. 24: [27] Отформатировано **Гергель** **05.10.2007 22:08:00**

русский

русский

Стр. 24: [27] Отформатировано	**Гергель**	**05.10.2007 22:08:00**

русский

Стр. 24: [27] Отформатировано	**Гергель**	**05.10.2007 22:08:00**

русский

Стр. 24: [27] Отформатировано	**Гергель**	**05.10.2007 22:08:00**

русский